META-CONTROL AND FUNCTIONAL RELATIONS IN LOGIC PROGRAMMING

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

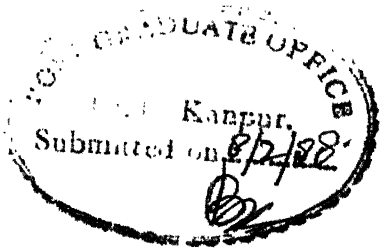
VINOD KUMAR DASHORA

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY, 1988



CERTIFICATE

This is to certify that the thesis entitled **Meta-Control And Functional Relations In Logic Programming** is a report of work carried out under our supervision by Vinod Kumar Dashora and that has not been submitted elsewhere for a degree.

Dr.H.Karnick	Dr. R. Sangal	Dr. T.V.Prabhaka
Asst. Professor	Asst. Professor	Asst. Professor
Dept. of C.S.E.	Dept. of C.S.E.	Dept. of C.S.E.
I.I.T.,Kanpur.	I.I.T.,Kanpur.	I.I.T., Kanpur.

Place: Kanpur
Date : February 1988.

ACKNOWLEDGEMENTS

It is a great pleasure to acknowledge my indebtness to my guides Dr. H. Karnick, Dr. R. Sangal and Dr. T.V. Prabhakar. I am particularly grateful to them for their constant encouragement. I am also thankful to all my friends who made my stay in IIT, Kanpur enjoyable.

V. K. Dashora

ABSTRACT

A meta-level facility for control in a logic programming shell is implemented. The control information in the form of meta-knowledge essentially specifys how to use the object-level knowledge. With the help of this facility a user can define his strategies and heuristics separately from the program and achieve better performance of execution.

We have also looked at the problem of specifying functional relation information in logic programming. The notion of functionality is declarative and independent of any search-strategy used in the inference. We have developed an algorithm for the interpreter which makes use of functional information and eliminate unnecessary recomputations in the inference of a goal.

CONTENTS

1. INTRODUCTION

1.1 Logic Programming

Logic programming began in early seventies when Kowalski [K74] gave the idea of using logic as a language for programming. It has an abstract underlying model viz. logic which provides a precise language for ones goal, knowledge, and assumptions. Ideally, logic programming only requires that knowledge about the problem and the assumptions which are sufficient to solve the problem be stated explictly as logical axioms. The logic program then can be executed by providing it with a problem i.e. a goal statement that is to be inferred. The execution of the program is an attempt to infer the goal statement given the assumptions in the logic program. This makes the logic program simple and declarative.

As pointed out by Kowalski, logic programs besides being declarative in nature have a procedural interpretation too. A program clause i.e.

A <- B1, B2, ... ,Bk.

can be regarded as a definition of procedure A. Further if

<- G1, G2, ... ,Gn

is a given goal clause then each of Gj is regarded as a procedure call. Procedural reading of the program clause will look like " goal A is true if goal G1 and goal G2 and ....and goal Gn are true. " Thus procedural interpretation of logic together with its declarative nature makes it a

very powerful language.

1.2 Language For Logic Programming

1.2.1 Prolog

The idea of using first order logic or at least a substantial subset of it as a programming language was quite promising which resulted in the design of the language Prolog [CM84] [SS86]. It is based on the Horn Clause subset of the logic. Every program clause whether it is representing knowledge for solving the problem or goal, has to be a Horn Clause. In a program clause i.e.

A <- B1, B2, ... , Bn.

A is called as head or consequent of the clause and B1,...,Bn is called body or antecedent of the clause. If the body of the clause is absent then its a fact which is always true and if the head is absent then its a goal clause. When both head and body are present then its a rule.

1.2.2 Prolog Interpreter

The prolog program, which is a set of Horn Clauses, is executed by specifying a goal

<- G1,...,Gn.

The interpreter starts execution by repeatedly executing inference cycles till the empty goal is reached. Each inference cycle takes a current goal to prove say

<- C1, ... , Ck.

It chooses the leftmost literal in the goal i.e. C1 and searches the database of program clauses to choose the first clause say

A <- B1, ... ,Bn.

such that the head of the clause i.e. A unifies with the chosen literal with the substitution s. The body of the clause B1,...,Bn is substituted in place of the chosen literal C1 and the substitution s is applied which results in the new goal

(B1, ... ,Bn,C2, ... Ck) s

to be solved. The following example illustrates this behaviour

Example 1_1 : Let us consider an example of family data-base with predicates defining parent, male, female and mother relations.

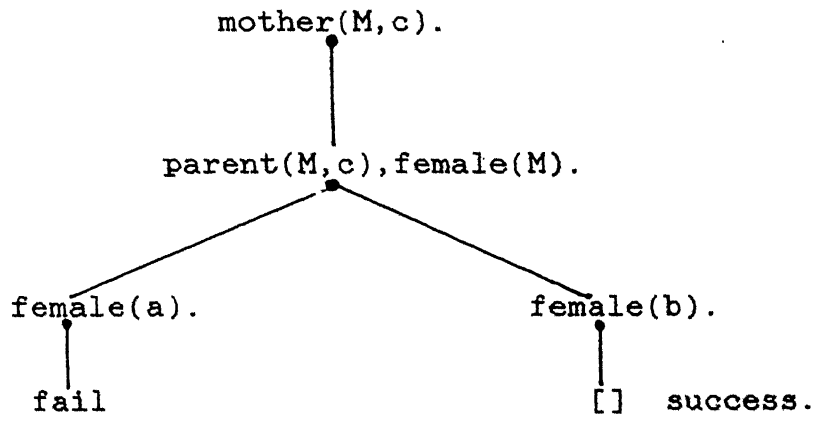
```
parent(a,e).          male(a).
parent(b,f).          male(c).
parent(a,c).          female(d).
parent(b,c).          female(b).
parent(c,d).

mother(X,Y) <- parent(X,Y),female(X).
```

Let the goal specified be mother(M,c), to know who is the mother of c. The goal has only one literal so its chosen. The literal chosen matches with the head of the rule defining mother relation. All the variables in the rule chosen are renamed so that they are different from all the variables appearing in the goal. The literal chosen unifies with the head of the rule and gives the substitution s {M/X,c/Y} which results in

parent(M,c),female(M).

the new goal to be solved. The complete trace of the proof can be seen in terms of the proof tree.



Each node in the proof-tree represents the goal or resolvent that is to be proved and each branch shows the clause used for proving the chosen literal. If a node is empty it shows success. Whenever a failure node is encountered the interpreter moves to the immediate ancestor of the failure node and takes the next branch. When the inference is proceeding successfully, it is moving down the proof-tree, then it is said to be moving in the forward direction. Whereas if failure is encountered at any node, it moves up the proof-tree and tries the next branch of the proof-tree, then it is said to be backtracking.

The inference-strategy embeded in the interpreter can be seen in terms of three decisions which the interpreter makes in proving the given goal.

(1) At any node in the proof-tree a literal is to be chosen for resolution. In Prolog the interpreter chooses the leftmost literal.

(2) At a node in the proof-tree, after the choice of literal is made, the interpreter has to choose a branch

for going to next node in the tree. This choice correspond to the choice of clause whose head is to be unified with the literal.The interpreter takes the first branch which correspond to the first clause in the order in which they are defined in the program.

(3) In case of failure at a node, it has to try alternate branches of the proof-tree. The interpreter goes back to the immediate ancestor and takes next branch in order until all possibilities for a literal are ultimately exhausted, then it fails.

The inference-strategy described performs the search for a solution of the goal in the search-space in a depth-first manner. Unification of a literal with the head of a clause performs basic data manipulation operations of variable assignment, parameter passing, data selection and data construction.

1.3 Problem Definition

In this thesis we have attempted two problems. One is the problem of control in logic programming and the other is the problem of specifying functional relation information in logic programming.

1.3.1 Problem Of Control

The problem of control can be seen in terms of directing the search for a solution in the search-space and avoiding searching useless paths. The problem of controlling the search essentially means determining which part of the

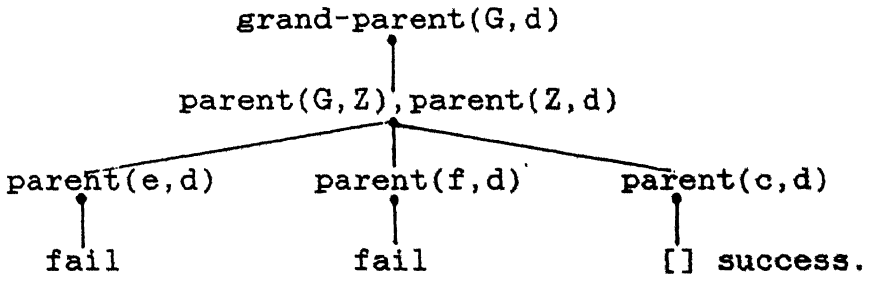
search-tree is useful to explore and in which order to expand the search-tree. This aspect of control concerns the search for a solution when inference is moving down the proof-tree i.e. in forward direction and is decided by the choice of literal in the resolvent at a node and the choice of the clause to be used in resolving the literal.

Ideally speaking an efficient forward control in terms of judicious choice of literal and clause may render backtracking unnecessary. This can be seen in the interpreter based on look-ahead of the literals [KM87]. The other aspect of control becomes important when failure is met at a node in the proof-tree. Inference moves up the proof-tree undoing the branches it has taken and tries the remaining branches systematically without taking into account the cause of failure. An efficient backtracking control may suggest which branch to undo. The need of an efficient control can be seen by the following example

Example 1_2 : In the example of the family database (Example 1_1) now we add one more rule for grand-parent relation as shown

grand-parent(X,Y) <- parent (X,Z),parent(Z,Y).

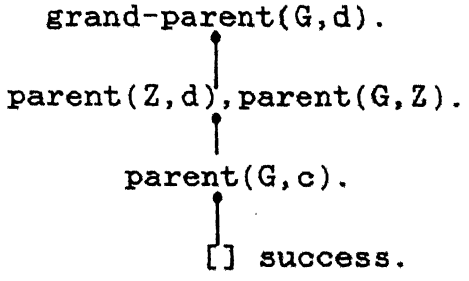
For finding out the grand-parent of d we execute the goal grand-parent(G,d). The trace of the proof is shown in the proof-tree



Now instead if we define the rule for grand-father with different order of literals in the antecedent as

grand-parent(X,Y) <- parent(Z,Y),parent(X,Z).

then the inference for the given goal will be faster as shown in the proof-tree



But with the latter definition of the grand-parent rule the same inefficiency will recur if we try to execute the goal for finding grand son of a i.e.grand-parent(a,S).

One of the main ideas of logic programming, due to Kowalski [K79], is that algorithm consists of two distinct components : the logic component and the control component. The logic component specifies the knowledge to be used in solving the problem, while how to use this knowledge constitutes the control component. The efficiency of an algorithm can usually be improved by improving the control component without changing the logic component. The logic / control duality is similar to knowledge/meta-knowledge duality in

problem-solving as suggested by Davis [D8Ø]. The main idea underlying this distinction is to separate information which expresses knowledge from the information which expresses how to use that knowledge. Therefore it would be very useful to device some way in which user can specify the control-information, in the form of meta-knowledge, separately from the program.

1.3.2 Functional Relation In Logic Programming

The second problem is the problem of specifying functional relations in logic programming. In general, predicates in the logic program define relations among its arguments. It is found that very often these relations are functional in nature. For example the predicate mother(X,Y) says that X is the mother of Y. There is a unique value of X for a given value of Y. There is no way of specifying such information describing functionality among the arguments of the predicate. Further, the logic programming interpreter, although very powerful due to its ability to simulate non-determinism, does not make use of such functional information while inferring the goal and ends up doing a lot of unnecessary recomputation. This is illustrated by the following example:

Example 1_3 : Let there be a goal <- g(X,Y),t(Y).The predicate g is such that there exists a relation between the argument X and Y which is functional in nature i.e. X uniquely determines Y. Let the goal be invoked with some

value of X. Further let the literal g(X,Y) succeed, getting some value for Y and literal t(Y) fail. Now the interpreter, which is unable to make use of the functional information, makes an unnecessary attempt to resatisfy the literal g(X,Y) for an alternate value of Y. If the interpreter is made such that it makes use of the functional information then such unnecessary recomputation can be avoided.

1.4 Aim Of This Work

In this thesis we have attempted solutions to both the problems described above. Our implementation would allow the user to improve the efficiency of his logic program written in VIDHI [S86], (a logic programming shell akin to prolog developed at IITK) by providing the control information pertinent to his program separately. Also, we have developed an interpreter which makes use of functional information provided by the user.

1.4.1 Meta-Control In Logic Programming

A meta-level control facility is proposed by Dincbas and Lepape in Metalog. We have developed a meta interpreter and implemented the core meta-level facilities proposed in Metalog.These facilities enable the user to describe forward control as well as control of backtracking in the inference. Moreover they also help the user to specify local control (specific to a particular literal, clause or resolvent) or global control (applicable to whole search-space) of his problem. With the help of these facilities, user can

specify control information pertinent to his program, separately, using the same syntactic formalism.

1.4.2 Functional Relation In Logic Programming

We have implemented a scheme for specifying functional information in a logic program. Further we have developed an algorithm for interpreter which makes use of the functional information and eliminates unnecessary recomputation. We have also shown the correctness of the algorithm.

2.META-CONTROL IN LOGIC PROGRAMMING

2.1 Introduction

The major reason for inefficiency of a prolog like proof-system is its pre-defined strategy which makes the proof-system highly inflexible and leaves the user with limited control over the search-space. To achieve efficiency the user should be allowed to control the order of trying out program-clauses and the order in which literals in the body of a clause can be chosen. This has to be included in the definition of the problem which makes it difficult to develop, understand and modify the program. Besides, the user has no control over backtracking as it is implicitly included in the interpreter. Thus due to pre-defined strategy, the user does not have any facility for expressing strategies, heuristics and control information specific to the program. To do this it is necessary to keep control outside the interpreter and give the user a facility to specify the control information externally.

2.2 External Control Combined With Program

It is possible to add some control primitives to a logic program, in the same syntactic framework, for achieveing control over some aspect of the inference process like selecting a literal, clause or backtracking point. Also these primitives make the logic program more deterministic and in the process achieve better performance. In PROLOG cut is an example of such a primitive. When selected it succeeds

and as a side effect commits the interpreter to the choice of the clause in which it occurs and all the choices made for the literals occuring before it in the clause. As a result subsequent backtracking behaviour is modified. Another example is annotation of variables in IC-PROLOG [CM79]. Using variable annotation one can define a different order of evaluation for different usage of a clause. This can also be used for achieveing delaying of a literal in a resolvent until one or more of its variables are bound.

Although using such control primitives increases the efficiency of inference it also makes the program less readable and difficult to change. Morevover these attempts do not take us closer to the ideals of logic programming where logic component and control component are separate.

2.3 External Control Separated From Program

Separating the control component from the logic component improves program clarity and makes it possible to specify and modify control of the program without altering the logic component. This separation allows the user to express control at a micro level: for example control could be specified for a particular clause or a particular literal (i.e. local strategy) alternately it could be global so that it is applicable to the whole search-space. The expression of control information in the form of meta-rules and meta-language was suggested by Davis [D8Ø] and Gallaire [GL82]. These ideas are further improved and made concrete in Meta-log proposed by Dincbas and Lepape [DL84]. We have imple-

mented the basic meta-level facilities given in Metalog for the logic programming shell VIDHI.

2.4 Organisation Of Meta-Level Control

In our implementation of the meta-control facility there are two distinct levels in which knowledge is expressed.

(1) **Object-Level** : Where object knowledge or domain specific knowledge for solving the problem is expressed. It represents the definition ('What' kind of knowledge) or logic component of the problem

(2) **Meta-Level** : Where meta-knowledge or control information about solving the problem is expressed. It represents the strategies or heuristics ('How to' kind of knowledge)to be used in applying the object knowledge for solving the problem.

Both object knowledge and meta knowledge are to be expressed in Horn Clauses using the same syntactic framework (a la prolog). Object knowledge is expressed by defining object-clauses (i.e. object-facts and object-rules) while meta knowledge is expressed by defining meta-clauses (i.e. meta-facts and meta-rules). A meta-clause is distinguished from an object-clause by the presence of a meta-predicate in the consequent position. A meta-predicate is an action predicate which is pre-defined in the system and whose semantics is known to the interpreter. The terms of meta-clauses are clauses, resolvents and literals of object-knowledge. Thus through meta-clauses the user has access to the entire set

of clauses and the complete proof-tree.

2.5 Operation Of Interpreter With Meta-Control

2.5.1 In Absence Of Meta-Clauses

In the absence of any meta-clause the interpreter chooses default strategy (i.e. the prolog strategy)

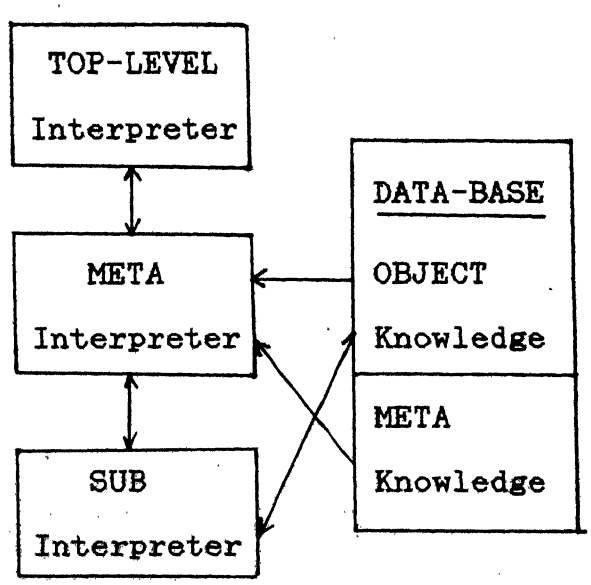
(1) Choosing left-most literal in the resolvent

(2) Choosing a clause in the textual order of clauses for resolution of the selected literal.

(3) In case of failure systematic backtracking to the last choice made and trying next alternative.

2.5.2 In Presence Of Meta-Clauses

In the presence of meta-clauses, control specified in the meta-knowledge is used in applying the object-knowledge to solve the problem. There are three main modules which can be identified in describing the operation of the interpreter with meta-control.



The top-level interpreter takes a goal that is to be inferred and activates the meta-interpreter at various decision points in the inference. Meta-interpreter when activated, analyses meta-clauses during which it invokes the sub-interpreter for processing of goals given in the antecedent of a meta-clause. When goals in the antecedent of a meta-clause succeed the interpreter acts according to to the meta-predicate present in the consequent position of the meta-clause. As a result the meta-interpreter constructs the strategy to be applied and the top-level interpreter applies it in inferring the goal. The sub-interpreter is a prolog like interpreter with a pre-defined strategy mainly used for sub-processing of the conditions in meta-clauses. The goals occuring in the antecedent of a meta-clause are object-goals which are either user defined or system defined.

2.5.3 Analysis Of Meta-Clauses

A meta-clause which has meta-knowledge embeded in it will essentially look like

(<meta-predicate> <- <cond1> <cond2> . . . <condn>).

Each meta-clause is charcterised by some meta-predicate in its consequent position. These meta-predicates which are action predicates should occur only in the consequent position and no where in the body of the meta-clause. Meta-clauses are analysed by the meta-interpreter in the order in which they are defined and depending upon the decision point attained by the top-level interpreter. The arguments

(i.e. terms) of the meta-predicates are clauses resolvents and literals from the object-knowledge. The types of the arguments of a meta-predicate are known to the interpreter along with the semantics of the meta-predicate. Once the condition on arguments of a meta-predicate is satisfied, evaluation of conditions in the antecedent of the meta-predicate begins. If the evaluation results in success then interpreter takes action corresponding to the meta-predicate. Whereas if it fails the next meta-clause is tried. When all the meta-clauses are exhausted then the default choice is taken.

2.5.4 Semantics Of Meta-Predicates

The naive interpreter (i.e. one without meta-control) has three main decision points :

(1) choice of the literal in a resolvent

(2) choice of clause for resolution of the selected literal and

(3) choice of backtracking point

The selection of meta-predicates has been influenced by our intention of extracting these choices from interpreter and making them available to the user at the meta-level. In addition meta-predicates are also provided to inhibit the choice of a literal, clause or backtracking point. Syntax and semantics of meta-predicates are as follow:

Selection Of Literal

(1) (activate <rslv> <lit>) <- <cond1> . . . <condn>

activates literal <lit> in the current resolvent <rslv> if conditions in the antecedent (i.e. <cond1> ... <condn>) are satisfied.

(2) (freeze <rslv> <lit>) <- <cond1> . . . <condn>

freezes a literal <lit> in the current resolvent <rslv> if conditions in the antecedent (i.e. <cond1> ..<condn>) are satisfied.

Selection Of Clause

(1) (chooseclause <lit> <body>) <- <cond1> . . . <condn>

chooses the clause, for rsolution of the selected literal <lit>, whose antecedent is charcterised by <body> if conditions in the antecedent of meta-clause (<cond1>..<condn>) are satisfied.

(2) (inhibclause <lit> <body>) <- <cond1> . . . <condn>

inhibits the choice of a clause, for resolution of the selected literal <lit>, whose antecedent is charcterised by <body> if the conditions in the antecedent of the meta-clause (<cond1>..<condn>) are satisfied.

Selection Of Backtracking Point

(1) (backfail <lit> <rslv>) <- <cond1> . . . <condn>

chooses a backtracking resolvent <rslv> in case of failure on literal <lit> if conditions in the antecedent of the meta-clause are satisfied.

(2) (inhibback <lit> <rslv>) <- <cond1> . . . <condn>

inhibits the choice of backtracking resolvent <rslv> in

case of failure on literal <lit> if the conditions in the antecedent of the meta-clause are satisfied.

In the meta-clauses as shown above the literals in the antecedent of a meta-clause are optional. In case they are absent it defines a meta-fact.

2.6 Case Studies

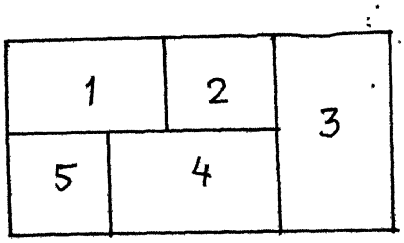
By judicious choice of control strategies the user can achieve significant improvement in the efficiency of inference. We give here a few examples of meta-programming and describe some of the useful control strategies we have used. The user can refer to the VIDHI manual [S86] for more details on syntax for defining clauses. It also has facilities for defining computational-predicates, question-rules (ask-user) and procpred etc.

Example 2_1 A map is to be coloured with at most four colours such that no two

adjacent areas of the map have the same colour.

object-clauses

```
(defassert ex1-12 (colour ?x1 ?x2 ?x3 ?x4 ?x5)
                  <-
                   (nxt ?x1 ?x2)
                   (nxt ?x1 ?x4)
                   (nxt ?x1 ?x5)
                   (nxt ?x2 ?x3)
                   (nxt ?x2 ?x4)
                   (nxt ?x3 ?x4)
                   (nxt ?x4 ?x5))
```




```
(defassert ex1-1  (nxt red blue))
(defassert ex1-2  (nxt red green))
(defassert ex1-3  (nxt red yellow))
```

```
(defassert ex1-3  (nxt blue red))
(defassert ex1-4  (nxt blue green))
(defassert ex1-5  (nxt blue yellow))
(defassert ex1-6  (nxt green red))
(defassert ex1-7  (nxt green blue))
(defassert ex1-8  (nxt green yellow))
(defassert ex1-9  (nxt yellow red))
(defassert ex1-1Ø (nxt yellow green))
(defassert ex1-11 (nxt yellow blue))
```

The object-facts in the database describe the occurence of colour in the adjacent areas. The object-rule defines the given graph.

We have tried two control regimes and found it improves the performance in terms of number of resolutions carried out. The strategies which were imposed through meta-control were the following :

Early Detection Of Failure

The selection of literal to be resolved in the resolvent should be such that the failures are detected quickly. If there is a literal which would definitely fail then choosing other literals merely delays this inevitable failure. This idea can be incorporated in the strategy of choosing a literal first which is totally instantiated, as choosing other literals will not change it. This strategy is expressed in the following meta-rule

```
(defassert mex2-1 (activate ?r (nxt ?x ?y))
                  <- (totalinst (nxt ?x ?y)))
```

It chooses a literal (nxt ?x ?y) in any resolvent if it is a totally instantiated literal. The predicate totalinst is a system predicate defined using defproc facility in VIDHI [S86].

Intelligent Backtracking

The other strategy which we have tried is a form of intelligent backtracking. In this when failure is encountered then instead of backtracking naively, backtracking is done to the resolvent which does not contain any instance of failed literal. This can be expressed by the meta-rule

```
(defassert mex2-2 (backfail ?p ?r)
                  <- (not (in ?p1 ?r) (subsume ?p ?p1)))
(defassert i1 (in ?x (?x . ?y)))
(defassert i2 (in ?x (?y . ?z)) <- (in ?x ?z))
```

The literal subsume in the antecedent of meta-rule is defined using defproc facility in Vidhi. A complete trace of the deduction using these meta-rules is given in the appendix .

performance of meta-control

GOAL : (colour ?x1 ?x2 ?x3 ?x4 ?x5)
R : no. of successful resolutions
B : no. of backtracking points.

	without meta-control	with meta-control
		(mex2-1)
R	20	9
B	13	1
		(mex2-2)
R	20	11
B	13	1

Example 2_2 This is the problem of reversing a list

object-clauses

```
(defassert ex2_1 (rev nil nil))
(defassert ex2_2 (rev (?x . ?y) ?z)
                 <- (rev ?y ?t) (app ?t (?x) ?z))

(defassert ex2_3 (app nil ?x ?x))
(defassert ex2_4 (app (?x . ?xs) ?y (?x . ?zs))
                 <- (app ?xs ?y ?zs))
```

We observe that whenever the first argument of the literal rev is a variable and the second one is instantiated (i.e. (rev ?x (1 2 3))) then ex2_2 is applied and it generates the rev goal with both arguments as variables. These are bound to nil and then app goal is tried. If app does not succeed then it backtracks and resatisfies the goal rev. This way it keeps going on till the length of first argument in the original rev goal becomes equal to the length of the second argument. This unnecessary backtracking can be cut down by activating app goal in the resolvent before the rev goal. This idea can be expressed by either of the following meta-clauses :

```
(defassert mex2_1 (activate ?r (app ?x ?y ?z)))
(defassert mex2_2 (freeze ?r (rev ?x ?y))
                  <- (var ?x))
```

performance of meta-control

GOAL : (rev ?x (1 2 3))
R : no. of successful resolutions made
B : no. of backtracking points.

	without meta-control	with meta-control
R	14	1Ø
B	4	Ø

Example 2_3 This is the problem of finding whether two trees have same sequence of leaf-tags.

object-clauses

```
(defassert ex3-1 (same-leaves ?t1 ?t2)
                 <- (leaves ?t1 ?l)
                    (leaves ?t2 ?l))
(defassert ex3-2 (leaves (leaf ?x) (?x)))
(defassert ex3-3 (leaves (tree (leaf ?x) ?t) (?x . ?l))
                 <- (leaves ?t ?l))
(defassert ex3-4 (leaves (tree (tree ?ll ?lr) ?r) ?l)
                 <- (leaves (tree ?ll (tree ?lr ?r)) ?l))
```

To find whether two trees have the same sequences of leaf-tags, goal (same-leaves <tree1> <tree2>) is executed. It matches with the rule for same-leaves and generates two leaves literals. The first leaves literal generates leaf-tags for the first tree and second one tests whether the tags generated are same as those of the second tree. In case of failure, generation of complete sequence of leaf-tags by first leaves literal is felt unnecessary. Instead, if the tags are tested as and when they are generated rather than

waiting for the complete generation, failure will be detected much faster. This is achieved by having co-routining between the two leaves literals. The following meta-control expresses this strategy :

```
(defassert mex3-1 (activate ?mr (leaves ?mx (?mu . ?mv)))
                        <- (atom ?mu))
```

performance of meta-control

GOAL : (same-leaves (tree (leaf a) (leaf b))
(tree (leaf b) (leaf a)))
R : no. of successful resolutions
B : no. of backtracking point

	without meta-control	with meta-control
R	3	2
B	4	3

Example 2_4 Consider a family databse describing father and grand-father relations.

object-clauses

```
(defassert a1 (father a b))
(defassert a2 (father a c))
(defassert a3 (father b d))
(defassert a4 (father b e))
(defassert a5 (father d h))
(defassert a6 (father d i))
(defassert a7 (father c f))
(defassert a8 (father c g))
(defassert a9 (father f j))
(defassert a1Ø (father f k))
(defassert r1 (gr-father ?x ?y) <- (father ?x ?z) (father ?z ?y))
```

As we have seen in the chapter 1 how the gr-father goal in which first argument is a variable generates unnecessary failure branches. To overcome this we need to change the order of literals in the rule for gr-father such that it does not affect the computation of other goal. This is expressed in either of the following meta-clauses :

```
(defassert m1 (freeze ?r (father ?x ?y))
                      <- (var ?x) (var ?y))
(defassert m2 (activate ?r (father ?x ?y))
                      <- (atom ?y))
```

performance of meta-control

GOAL : (gr-father ?x k)
R : no. of successful resolution
B : no. of backtracking points

	without meta-control	with meta-control
R	9	3
B	6	Ø

Remark

In all the examples meta-control gives better performance in terms of reduced number of resolution and backtracking points. However, improvement in performance is at the cost of certain overheads involved in sub-processing of condition in meta-clauses. At every decision point meta-clauses are analysed and goals in the body of meta-clauses are satisfied. Thus meta-control cuts down the search-space of the problem by introducing search in the search-space of

sub-problem. The meta-control is specially advantageous when serch-space of the sub-problem is smaller than that of the problem which is to be solved.

2.7 Implementation

Implementation of the meta-control facility was done in clisp environment on top of Franz Lisp.

2.7.1 Representation Of Database

The representation scheme for object-knowledge database and meta-knowledge database is same. Syatax for defining a clause (object-level or meta-level) is as

(defassert <cls-name> <consequent> <- <antec1> ... <antecn>)

which is same as the syntax for defining a clause in Vidhi. The literals in consequent and antecedent of the clause look like

(<predicate-name> <arg1> <arg2> . . . <argn>).

Object facts and object rules are stored as property of the atom <predicate-name> under the attributes facts and rules respectively. Meta-clauses are stored as property of the atom $meta under the attributes $chlit, $chcls and $bt depending on which meta-predicate is occuring in the meta-clause.

2.7.2 Representation Of The Environment

The environment is stored as an association list of variables and their bindings. All the variables in the clause are renamed when the clause is selected for resolving the chosen literal. Renaming of variables ensures that variables in the clause are different from those occuring in the

goal. Each entry in the stack, which corresponds to one inference cycle in the deduction of the goal, contains an association list of variables instantiated till that step.

2.7.3 Inference

2.7.3.1 Data Structure

The data structure chosen for implementing the inference mechanism in the interpreter with meta-control is the stack. An iterative algorithm for inference is implemented so that at any step the complete proof-tree is accessible. Each entry in the stack corresponds to a activation record containing object-information (obj-info) and meta-information (meta-info).

Obj-info is alist of five elements which looks like :

(<query> <qlist> <facts> <rules> <alist>)

where <query> is the literal chosen in any step from the current resolvent <qlist>. The <facts> and <rules> are the facts and the rules that remain to be applied for solving the literal <query>. The last element in the obj-info is the association list of variables till that step. At any time when inference is proceeding in the forward direction, the entries corresponding to <query>, <facts> and <rules>, in the top activation record, will be $query $facts $rules respectively. As and when the decisions about the choice of literal and clause are made, the entries $query, $facts and $rules are suitably substituted.

The second entry in the activation record includes the information about meta-clauses (i.e. meta-info). It is a

list of three element which looks like

```
( meta-clauses-for-choosing-literal
  meta-clauses-for-choosing-clause
  meta-clauses-for-choosing-bt-point )
```

Depending upon which decision is to be made in the interpreter, the corresponding meta-clauses are analysed and meta-information is updated, so that on backtracking the remaining meta-clauses can be tried.

2.7.3.2 Data Structure Definition

Each entry in the stack as described above is defined as a Lisp Structure. It is advantageous to do so as it provides appropriate definition of constructor and selector function.

```
(defstruct (entry (:constructor make-entry (obj-info
                                            meta-info)))
           obj-info
           meta-info )
(defstruct (obj-info (:constructor make-obj-info (query
                                                  qlist
                                                  facts
                                                  rules
                                                  alist)))
           query
           qlist
           facts
           rules
           alist)
(defstruct (meta-info (:constructor make-meta-info
                                    (chlit-rules
                                     chcls-rules
                                     bt-rules )))
           chlit-rules
           chcls-rules
           bt-rules)
```

2.7.3.3 Algorithm For Meta-Control

INPUT : Input to the interpreter is a goal (list of literals) i.e. qlist that is to be solved.

```
    ;* qlist is renamed and literal (next) is to be added
    ;* at the end./
    ;* initialisation of stack /
    ;* in the begining alist is nil /

  (let ((ini-stack (make-entry (make-obj-info '$query
                                              qlist
                                              '$facts
                                              '$rules
                                              nil)
                               (make-meta-info
                                     (get-chlit-rules)
                                     (get-chcls-rules)
                                     (get-bt-rules)))))
        ;* get-chlit-rules, get-chcls-rules, get-bt-rules
        ;* are functions for getting meta-clauses for
        ;* choosing literal, clause and backtracking
        ;* point respectively.
    ;* TOP-LEVEL ITERATION
    ;* initialisation of iteration variables
       (do* ((stack ini-stack)
             (top-stack (car stack) (car stack))
             (obj-info-lcl (entry-obj-info top-stack)
                           (entry-obj-info top-stack))
             (meta-info-lcl (entry-meta-info top-stack)
                            (entry-meta-info top-stack))
             (qlist-lcl (obj-info-qlist obj-info-lcl)
                        (obj-info-qlist obj-info-lcl))
             (alist-lcl (obj-info-alist obj-info-lcl)
                        (obj-info-alist obj-info-lcl))
             (over-flag nil))

            ((or (null stack) over-flag)
             ;* Termination Condition for Top-Level
             ;* iteration.

             (cond (over-flag t)
                   (t '$fail-infer)))
     ;* Body of the TOP-Level iteration begins here /
     (cond ((null qlist-lcl)
            ;* sub-problem is solved./
            ;* This arises when the argument of negation
            ;* is succeeded. /
            (setq over-flag t))
```

```
(t
;* Inference cycle with the top-stack begins here/
;* First decision that is to be made is about the
;* choice of literal in the resolvent /
;* This choice is exclusive and not reconsidered
;* upon backtracking /
;* It calls meta-interpreter for choice of literal
;* The argument given to it are obj-info and
;* meta-info /

(apply-chlit-rules obj-info-lcl meta-info-lcl)

;* It analyses meta-clauses for choosing literal
;* in the resolvent /
;* The chosen literal is substituted in place of
;* query /

(let (( query (obj-info-query obj-info-lcl)))

     ;* solve the query /
     ;* If query succeed then update top-stack
     ;* and create new top-stack /
     ;* If query fails then call meta-interpreter
     ;* for choice of backtracking point /

  (cond ((it-is-a-non-database-query)

         ;* Its a non-database query /
         ;* It could be any one of these
         ;* not, =, ask-user, defcomp, defproc,
         ;* cut, next  etc.

         (let ((res (solve-the-query query)))
              (cond ((not (fail-p res))

                     ;* query succeeds /

                     (modify top-stack))
                    (t

                     ;* query has failed /
                     ;* call meta-interpreter for
                     ;* choosing bt point /

                     (setq stack (apply-bt-rules stack))))))

        (t

;* Its a database query /
;* Call meta-interpreter for choosing a
;* clause for resolving the literal query /

;* ITERATION for choosing a clause /

        ;* Initialisation of iteration variables

(do ((clause (rename-vars (apply-chcls-rules obj-info-lcl
                                             meta-info-lcl))
             (rename-vars (apply-chcls-rules obj-info-lcl
```

```
                                                meta-info-lcl)))
    (lcl-res (unify query
                    (asrt-conseq clause)
                    alist-lcl)
             (unify query
                    (asrt-conseq clause)
                    alist-lcl)))
            ((null clause)
             ;* TERMINATION condition for iteration /
             ;* All facts and rules are exhausted so
             ;* call meta-interpreter for choosing
             ;* a backtracking resolvent /

            (setq stack (apply-bt-rules stack)))
            ;* BODY of Iteration for choosing a clause
           (cond ((fail-p lcl-res)
                  ;* Chosen clause did not unify /
                  'skip-and-try-other-clause )
                 (t
                  ;* Chosen clause has unified /
                  (modify-current-top-stack)
                  (create-new-top-stack)
                  (return)))))))))))
```

2.8 Conclusion

We have implemented meta-control facility and found that the combination of object and meta level (i.e. knowledge and meta-knowledge) gives to a system more natural and powerful control expression facility. Through the control specified at meta-level, the user can define his own strategy and heuristics explicitly and more elegantly. The powerful set of meta-clauses allows user to intervene on the process of deduction and guide the search for a solution. As a result better performance is achieved which is evident by the examples.

3. FUNCTIONAL RELATIONS IN LOGIC PROGRAMMING

3.1 Functional Relations In Logic Programs

In general, a predicate in a logic program describes the relation among its arguments. It is found that very often these relations are functional in nature. For example consider the predicate mother with two arguments mother(X,Y) describing the relation " X is the mother of Y ". We say that first argument X is the function of the second argument Y as Y uniquely determines X. Thus the arguments of predicate mother are functionally related. Formally a functional relation f is said to exist among the arguments of a predicate p, if a set of arguments in positions I of p uniquely determines a set of arguments in positions D of p. It is described as

f : I -> D for predicate p.

Further the predicate p is termed as a predicate with functional dependency or fd-predicate and sets I and D are known as the set of independent and dependent arguments of p respectively.

In a logic program P an fd-predicate say mother(X,Y) (where Y is the independent and X the dependent argument) suggests that there do not exist three terms u , v and t such that v $\neq$ t and mother(v,u) and mother(t,u) both are logical consequences of P. In addition to this, the functional relation information (fd-information) corresponding to an fd-predicate is applicable to the whole search-space

of the inference irrespective of the search-strategy used. Hence the specification of fd-information is global to the database and does not disturb the declarative nature of the logic program.

3.2 Functional-Computation in Logic Programming

3.2.1 Functionality vs Determinacy

The notion of functionality in a logic program is quite close to the notion of determinacy. Every deterministic computation is essentially functional in nature. The notion of deterministic computation suggests that it would succeed at most once giving a unique answer. But the notion of functionality says that it would always give unique answer(s) though it may succeed more than once. The example which distinguishes these two notion is given as follows

Example 3_1 Consider the following clauses

```
t(X,a).
t(X,Y) <- t(X,Y).
```

Here the predicate t is an fd-predicate in which the first argument uniquely determines the second. For a given value of first argument it computes a unique second argument. This computation although functional, for it produces a unique answer, is not deterministic as it can produce this answer infinite number of times.

The notion of determinacy is described by Nakamura [N86] at a logical and an operational level. Our notion of functionality is same as that of logical determinacy.

3.2.2 Deterministic-Computation in Logic Programming

The main purpose of making a logic program deterministic is to reduce the search-space for the computation of a goal. This improves the performance in terms of execution time of the goal and space requirement on the run time stack of the interpreter. The main tool available for making a logic program deterministic is cut (!). This is used to prune the fruitless paths in the search-space. When selected it succeeds and commits the interpreter to all choices made since the parent goal is unified with the head of the clause in which it occurs. For more elaboration on operational details of cut one may refer [CM84], [SS86]. The computation described in Example 3_1 can be made deterministic by introducing cut in the first clause as

```
t(X,a) <- !.
t(X,Y) <- t(X,Y).
```

Now the program produces unique answer for a given value of first argument only once.

3.2.3 Cut And Functional-Computations

The cut, although very powerful in its ability to prune search-space, offers some disadvantages as many of its uses can only be interpreted procedurally [SS86]. It is assymmetric as it prunes the clauses occuring below the clause in which it occurs and not the clauses above it. Moreover it affects the behaviour of clause in which it occurs irrespective of how the clause is being used in the inference. This

would become clear in the following example.

Example 3_2 Consider a logic program for appending two lists.

```
append([],Y,Y).
append([X|Xs],Y,[X|Zs]) <- append(Xs,Y,Zs).
```

Consider the goal append([1,2],[3,4],Answer). The computation performed by above goal is functional in nature as Y can be determined uniquely. But it is not deterministic because upon backtracking it would try second rule which remains to be applied. To overcome this we make it deterministic by introducing cut as

```
append([],Y,Y) <- !.
append([X|Xs],Y,[X|Zs]) <- append(Xs,Y,Zs).
```

This no doubt solves the above problem but introduces another problem. Now the goal apppend(X,Y,[1,2]) which is not functional at all also has a deterministic solution i.e.

X = [] , Y = [1,2]

Although, using cut, enables functional-computation in a logic program, it may make other computations deterministic which are not functional in nature. Thus cut can not handle the problem of functional computations. This problem is mainly due to cut occuring in the clause for making the computation deterministic. On the other hand functional relation information is specified for a predicate and it is global and declarative in nature.

3.3 Interpreter With Fd-Information

As discussed in the previous section, cut does not provide a right solution to the problem of performing functional-computation deterministically. The existing logic programming framework, although very powerful in its ability to simulate non-determinism, is found to be quite expensive when it comes to the problem of performing functional-computation. Moreover there is no way of specifying fd-information relevant to a logic program. We offer a solution to the above problem by proposing a logic programming framework in which

(1) it would be possible to specify functional relation information in terms of fd-predicates in the program.

(2) the interpreter would make use of such functional relation information and eliminate unnecessary recomputation by making the computation deterministic according to the functional relation information.

3.3.1 Terminology

In this section we introduce some terminology which we have used in describing our algorithm and its implementation. For details on syntax of defining clauses and posing query etc. one may refer to VIDHI [S86]. A summary of basic commands is also given in APPENDIX 1.

The functional relation information corresponding to a fd-predicate can be specified in Lisp like syntax as

(deffd (<pred-name> <arg1> <arg2> ... <argn>))

where <argi> for 1 <= i <= n is any of the following

i : if <argi> is an independent argument

d : if <argi> is a dependent argument

x : if <argi> is neither an independent nor a dependent argument (i.e. a dontcare argument)

The fd-information corresponding to the fd-predicate (temp ?pos ?temp), where the first argument ?pos uniquely determines the second argument ?temp, can be represented as

(deffd (temp i d)).

At any time during the inference, if there is a literal which is an fd-predicate then a variable occupying independent/dependent/dontcare argument position of the literal is known as an independent/dependent/dontcare variable w.r.t. some functional relation.

For making use of the functional relation in the inference, it is important to determine whether a particular functional relation is applicable to the computation of a goal. If all the independent argument position w.r.t. the functional relation in the literal are ground terms then that functional relation is said to be applicable to the literal.

A goal literal p is said to succeed totally if either

(1) there is a fact q in the database such that p unifies with q or

(2) there exists a rule r in the database such that p unifies with head of the rule and all literals in the antecedent of r succeed totally.

As the inference proceeds, the relevant fd-information is accessed and stored as passive-dependency-information (passive-dpd-info) and active-dependency-information (active-dpd-info). The functional information is usually passed through the passive-dpd-info into active-dpd-info. The functional information in passive-dpd-info and active-dpd-info is kept in the form of a pair of list of dependent arguments and corresponding fd-predicate. In an inference cycle, if a literal selected for resolving, say p, is an fd-predicate then fd-information applicable to that instance of the literal is added to the passive-dpd-info present in the system. When the literal p succeeds totally then all the fd-information corresponding to p present in passive-dpd-info is moved to active-dpd-info present in the system. At any time, all the information present in the active-dpd-info is active in the system and affects the computation of subsequent goals.

The presence of functional-relation information modifies the backtracking behaviour of the interpreter and as a result helps in eliminating unnecessary recomputation of the goal. When there are no functional relations specified in the program then in the event of failure the interpreter

performs naive backtracking which means going back to the last decision point in the search-tree and re-satisfying the goal by taking next alternative. Whereas, if fd-information is present then a literal may be restricted against backtracking. If a literal is restricted against backtracking then during backtracking the literal is not re-satisfied by trying next alternative. The idea of restricting a literal against backtracking can be illustrated as follows :

Consider a resolvent R, at any time in the computation of a goal as

R : $<lit_1>$ .. $<lit_{i-1}>$,$<lit_i>$,$<lit_{i+1}>$, ..$<lit_k>$

Suppose first i-1 literals succeed and literal $<lit_i>$ is restricted against backtracking then in the event of future failure and subsequent backtracking the literal $<lit_i>$ is not re-satisfied.

In every inference cycle, a literal is chosen from the resolvent. The chosen literal is unified with the head of the clause for the literal. If the unification succeeds then it returns an association list containing the updated bindings of the variables. If the unification fails then it does not merely return the failure but also returns the argument number of the literal where failure has occured and the variable (if any) due to which failure is occuring.

In case of failure on a literal, active-fd-info is considered in making a decision about backtracking-point. This idea will become clear by the following illustration :

Let there be a resolvent R given as

R : <lit_1>..<lit_j>..<lit_i> .. <lit_k>

Suppose, as the inference procceds, literals <lit_1> to <lit_{j-1}> are succeeded totally. Let a literal <lit_j> be an fd-predicate with some functional relation applicable to it. If literal <lit_j> unifies with head of some clause then the corresponding functional relation information is added to passive-dpd-info present in the system. Further, if it succeeds totally then the corresponding functional information is moved from passive-dpd-info to active-dpd-info.Now latter, if literal <lit_i> fails such that failure is over some of the dependent arguments of the literal <lit_j> then instead of backtracking to the literal <lit_{i-1}> it backtracks to the literal <lit_{j-1}>. This is what is known as setting the backtracking point. If the failure of the literal is not over any dependent argument in active-dpd-info then backtracking is as usual.

3.3.2 Algorithm For The Interpreter

The algorithm for the interpreter using functional relation information is given in simple terms as follows:

Data Structure

The interpreter uses a stack. Each entry in the stack corresponds to one inference cycle containing the following information :

qlist : it is the resolvent containing literals which remain to be proved.

facts : all the facts corresponding to the first literal in the qlist.

rules : all the rules corresponding to the first literal in the qlist.

alist : association list containing the bindings of the variables till that point in the inference.

a-fd-info : association list containing active functional relation information and corresponding fd-predicates.

p-fd-info : association list containing passive functional relation information and corresponding fd-predicates.

In each inference cycle, first literal in the resolvent is solved. If success occurs then information on the top-stack is updated and new top-stack is created which is pushed on the stack. If failure occurs then top-stack is popped out and inference is continued.

INPUT : goal which is a list of literals to be solved i.e. qlist.

Initialisation Of Stack

```
top-stack <- ( qlist
               (all-facts-for (first-of qlist))
               (all-rules-for (first-of qlist))
               nil
               nil
               nil )

stack <- (top-stack)
```

```
Top-Level Iteration
;* for one inference cycle it goes through the loop
;* once.
KEEP REPEATING
1. If stack is empty then return failure
2. If qlist is empty then
      {
      2.1 print the answer
      2.2 if (more answer required) then
            {
            (pop the stack)
            (top-stack <- (first-of stack))
            }
          else
            {
            return
            }
      }
    else
       ;* qlist is non-empty
       ;* carry out the inference
       {
       ;* ITERATION for CHOOSING A CLAUSE
       ;* Initialisation
        query         <- (first-of qlist)
        qlist         <- resolvent or goal to be solved
                         in the activation record on
                         top of the stack.
        rules-to-try <- clauses for the query
        rule          <- (first-of rules-to-try)
       ;* If the head of the clause unifys with the
       ;* query then backtracking over query in case
       ;* of future failure is decided.
       ;* If unification fails then backtracking
       ;* point is set and it is decided
       ;* whether to try next available rule
       ;* or backtrack.
       if (query unifies with the
             head of the rule) then
         {
         if (and (query is an fd-predicate)
                 (there is a non-nil fd-relation
                  which can be applied to the
                  query)) then
           {
           if (unification instantiates only
               dependent arguments) then
              {
              1.restrict query against backtracking
              2.create new top-stack
```

```
        3.push new top-stack
        4.top-stack <- (first-of stack)
        }
     else
        ;* does not instantiate only dependent
        ;* argument.
        {
        ;* usual backtracking
        1.create new top-stack
        2.push new top-stack
        3.top-stack <- (first-of stack)
        }
    }
  else
     ;* query is not an fd-predicate
     ;* or fd-relation is nil
     ;* usual backtracking
     {
     1.create new top-stack
     2. push new top-stack
     3.top-stack <- (first-of stack)
     }
  }
else
   ;* query does not unify with the head
   ;* of the rule./
   ;* if there are active dependencies
   ;* then set backtracking point
   ;* if query is an fd-predicate and
   ;* cause of failure is dpd-variable
   ;* or dpd-argument then check for
   ;* contradiction
   ;* if contradiction is found then
   ;* backtrack otherwise try-next-rule
   {
 1.if (active-dpd-info present) then
     {
     if (failure violates active-dpd-info)
        then
           {
           set backtracking point
           }
     }
 2.if (and (query is an fd-predicate)
           (there are non-nil dpd-arguments))
           then
              {
              if (failure due to dpd-argument or
                        dpd-variable) then
                 {
                 if (contradiction) then
```

```
                    {
                    BACKTRACK
                    }
                  else ;* not a contradiction
                    {
                    TRY-NEXT-RULE
                    }
                  }
                else
                  ;* failure not due to
                  ;* dpd-variable or dpd-argument
                  {
                  TRY-NEXT-RULE
                  }
              }
        else
            ;* not an fd-predicate
            {
            TRY-NEXT-RULE
            }

    }
}
```

Correctness-Result

The correctness of the algorithm described above is shown. The terminology used in the proof is the standard one and can be referred in [L84].Our intention in the proof has been to show that given a logic program, a goal computed by the algorithm is logical consequence of the program and in accordance with the fd-information specified by the user.

Theorem : Let P be a logic program and g a goal. Let interpreter be as specified in the algorithm. The O be the answer-substitution (it has bindings of some or all of the variables appearing in the goal) obtained in the computation of the goal.

Then gO is the logical consequence of P, O is consistent w.r.t. fd-information specified in the program (i.e. all dependent variables in O are uniquely bound) and algorithm gives all possible answer substitutions.

Proof : CASE 1

There are no functional dependencies specified in the system. This part of the proof is standard proof for the logic programming interpreter and can be referred in [L84].

Let g be the goal given as

$<- A_1, A_2, \ldots, A_k$ and

$O_1, O_2, \ldots, O_n$

be the sequence of substitution obtained in each inference cycle in the computation of the goal g.

To show that

$\forall ((A_1, \ldots, A_k)O_1, \ldots, O_n)$

is a logical consequence of P.

The proof is by induction on the length of proof-tree of g.

(1) suppose n = 1 (BASE CASE)

=> g is a goal of the form $<- A_1$ and

program has a unit clause of the form A <-

such that

$A_1 O_1 = A O_1$.

Since $A_1 O_1$ is an instance

of the unit clause of P, it follows that

$\forall (A_1 O_1)$ is a logical

consequence of P.

(2) (INDUCTION HYPOTHESIS)

Assume that result holds true when g is computed by the algorithm with the proof-tree length (n-1).

Suppose $O_1, \ldots, O_n$ be the sequence of substitutions used in the computation of g with proof length n.

Let A <- $B_1, \ldots, B_q$ (q >= o) be the first input clause and A_1 be the selected literal of g such that

A O_1 = A_1 O_1.

Then by induction hypothesis

$\forall$ ((B_1 ... B_q $A_2, \ldots, A_k$) O_1 ... O_n)

is the logical consequence of P.

If q = $\emptyset$ then it is trivial, as A_1 is an instance of fact in the database.

If q > $\emptyset$ then

$\forall$ ((B_1 ... B_q) O_1 ... O_n) is logical consequence of P.

=> $\forall$ (A O_1 ... O_n) is a logical consequence of P.

=> $\forall$ (A_1 O_1 ... O_n) is a logical consequence of P.

=> $\forall$ ((A_1... A_k) O_1 ... O_n)

is a logical consequence of P.

CASE 2

The fd-information is specified in the system.

Let g be the goal given as <- $A_1, \ldots A_k$ and $O_1, \ldots, O_n$ be the sequence of substitutions obtained in each inference cycle in the computation of the goal.

To show that

$\forall$ (($A_1 \ldots A_k$) $O_1 \ldots O_n$) is a logical consequence of P and substitution O is consistent w.r.t fd-information specified in the system.

The proof is by inducion on the length of proof-tree for the computation of goal g.

(1) Suppose n = 1 (BASE CASE)

=> g is the goal of the form <- A_1.

Let f be the fd-information applicable to the goal g and program P has unit clauses of the form

A_{p1} <-

A_{p2} <-

- - -

- - -

A_{pm} <-

- - -

- - -

A_{pj} <-

such that $A_1 O_1 = A_{pm} O_1$ for some m, 1<=m<=j

Remark : In the interpreter based on the algorithm clauses for a literal are chosen according to their

textual order.

=> There does not exist any substitution O_k such that

$A_1 O_k = A_{px} O_k$

for all x, 1 <= x <= m-1.

Further none of the clauses A_{px} (1 <= x <= m-1) in the process of failing, find that the goal <- A_1 has any information contrary to f (i.e. literal A and A_1 differ only in some dependent argument corresponding to f).

Remark : If there exists some contrary information then it would not try remaining clauses.

Since $A_{pm} O_1$ is an instance of the unit clause of P therefore $\forall (A_1 O_1)$ is a logical consequence of P.

INDUCTION HYPOTHESIS :

Let the result hold true when g is computed by the algorithm with the proof-tree length (n-1).

Suppose $O_1,..,O_n$ be the sequence of substitutions used in the computation of g with proof length n. Let A_1 be first literal selected and let f be the fd-information applicable to A_1. Let the program have the clauses for A_1 as follow :

A_{p1} <- $B_{p11},...,B_{p1q}$.

- - -

- - -

A_{pm} <- $B_{pm1},\ldots,B_{pmq}$.

- - -

- - -

A_{pj} <- $B_{pj1},\ldots,B_{pjq}$.

Remark : In the interpreter based on the algorithm clauses are chosen according to their textual order.

Let a clause chosen be

A_{pm} <- $B_{pm1},\ldots,B_{pmq}$

such that

$A_{pm}\ O_1 = A_1\ O_1$

This implies that for clauses A_{px} <- ...where $1 <= x <= m-1$, one of the following cases hold true

CASE 1

There does not exist a substitution O_x such that

$A_1\ O_x = A_{px}\ O_x$

(unification with the head of the clause fails).

But failure does not indicate that A_1 has information which is contrary to the fd-information f.

CASE 2

(Unification with the head of the clause is successful)

There may exist O_x such that

$A_{px}\ O_x = A_1\ O_x$

then either of the two cases would happen

(1) antecedent of the rule is not proved i.e.

$\forall (B_{px1} \ldots B_{pxq})\ O_x$ is not
logical consequence of P.

(2) There occurs a failure ahead i.e.

$(A_2 \ldots A_k)\ O_x \ldots O_2 .. O_n)$
is not logical consequence of P

When A_1 succeeds totally then f becomes active-fd-information in the system. Although there is failure ahead but it does not indicate that a literal A_1 for 2 <= 1 <= k (literal which is failing) has any information contrary to the fd-information f.

Thus by induction hypothesis

$\forall ((B_{pm1} .. B_{pmq} .. A_2 .. A_k)\ O_1\ O_n)$
is a logical consequence of P and O is consistent w.r.t. fd-information f.

=> If q > Ø then $\forall (B_{m1} \ldots B_{mq})\ O_1 \ldots O_n$
is a logical consequence of P

=> $(A_{pm}\ O_1 \ldots O_n)$ is a logical consequence of P.

=> $\forall (A_1\ O_1 \ldots O_n)$ is a logical consequence of P.

=> $\forall ((A_1,..,A_k)\ O_1 \ldots O_n)$
is a logical consequence of P and substitution is consistent w.r.t. fd-information specified in the system.

(1) antecedent of the rule is not proved i.e.

$\forall (B_{px1} \ldots B_{pxq})\ O_x$ is not
logical consequence of P.

(2) There occurs a failure ahead i.e.

$(A_2 \ldots A_k)\ O_x \ldots O_2 .. O_n)$
is not logical consequence of P

When A_1 succeeds totally then f becomes active-fd-information in the system. Although there is failure ahead but it does not indicate that a literal A_1 for 2 <= 1 <= k (literal which is failing) has any information contrary to the fd-information f.

Thus by induction hypothesis

$\forall ((B_{pm1} .. B_{pmq} .. A_2 .. A_k)\ O_1\ O_n)$
is a logical consequence of P and O is consistent w.r.t. fd-information f.

=> If q > Ø then $\forall (B_{m1} \ldots B_{mq})\ O_1 \ldots O_n$
is a logical consequence of P

=> $(A_{pm}\ O_1 \ldots O_n)$ is a logical consequence of P.

=> $\forall (A_1\ O_1 \ldots O_n)$ is a logical consequence of P.

=> $\forall ((A_1,..,A_k)\ O_1 \ldots O_n)$
is a logical consequence of P and substitution is consistent w.r.t. fd-information specified in the system.

3.4 Implementation

A facility for specifying fd-inormation and an interpreter which makes use of functional information in the inference process has been implemented in the clisp environment available on top of Franz Lisp on Unix.

3.4.1 Representation Of Database

The facts and rules corresponding to a logic program are defined using the facility for defining assertions i.e.

(defassert <asrt-name> <conseq> <- <antec1> ... <antecn>)

This is same as the syntax for defining assertion in VIDHI. For more details on syntax for defining assertions, computational predicates, and procpreds one may refer to manual for VIDHI [S86]. Each literal in the consequent and the antecedent of the clause looks like

(<pred-name> <arg1> .... <argn>)

The facts and rules for a predicate are stored as property of the atom <pred-name> under the attribute facts and rules respectively. For resolving a literal chosen in the resolvent in any inference cycle, first the facts are tried and then the rules. The facts and rules are stored in the order in which they are defined.

3.4.2 Representation Of Environment

The environment at any step in the inference, is stored in the form of an association list of variables and their bindings. All the variables in a clause are renamed when the

clause is chosen for resolving the selected literal so that variables in the clause are different from all the variables in the resolvent. The association list in any inference cycle contains the variables bound till that point in the inference.

3.4.3 Representation Of Fd-Information

The fd-information corresponding to a predicate can be specified as

(deffd (<pred-name> <arg1> ... <argn>))

where each <argi> for 1 <= i <= n could be any one of the following :

i : if <argi> is an independent argument

d : if <argi> is a dependent argument

x : if <argi> is neither an independent nor a dependent argument.

The fd-information so specified is stored in some internal representation for fd-information. The internal representation of the fd-information corresponding to an fd-predicate looks like :

```
(<no-of-args> (<list-of-i-args>
               <list-of-d-args>
               <list-of-x-args>))
```

where the first member of the list is number of arguments

dontcare arguments. The internal representation is stored as a property of the atom <pred-name> under the attribute $fd.

3.4.4 Compilation Of Fd-Information

After the user has defined clauses for solving his problem and specified relevent fd-information then complete fd-information is compiled. The compilation of fd-information essentially means deriving all possible functional relation among the arguments of a predicate given some functional relation by the user. For example user can specify two functional relation for a predicate foo as shown

```
(deffd (foo i d x))
(deffd (foo x i d))
```

then functional relation in which the first argument uniquely determines the third one can be derived (by transitivity). Thus for every subset of the set of arguments of each fd-predicate, the set of dependent variables is computed by taking closure w.r.t. fd-information specified for the predicate. This information is stored in trie-structure. The number of arguments of an fd-predicate along with the trie-structure containing fd-information is stored as property of the atom <pred-name> under the attribute $fd-trie. An outline of the algorithm for compilation of fd-information is given as follows

INPUT : list of fd-predicates

OUTPUT : Computes closure of the arguments w.r.t. func-

tional relations specified for an fd-predicate and stores it in a trie-structure.

ALGORITHM :

for every fd-predicate in input list do

1. Initialisation
   arg-no <- (no of arguments of fd-predicate)
   trie-list <- nil
   arg-set <- set of argument position of
              fd-predicate (with inceasing argument
              number)
   ;* a fd-predicate with two arguments will have
   arg-set as (Ø 1)

2. *for every sub-set SS of arg-set do
   2.1 closure-list <- nil
   2.2 compute the closure C w.r.t. fd-information
       given for the fd-predicate
   2.3 add list of sub-set SS and corresponding
       closure C to closure-list

3. Construct a trie-structure T corresponding to closure-list. The first element of each entry in the closure-list acts as a key under which the second element is stored in the trie-structure.
4. Add an entry (<arg-no> T) to trie-list for the fd-predicate.
5. Store the trie-list as a property of the

atom <pred-name> under the attribute $fd-trie.

The compilation of fd-information and storing in the trie-structure makes it easy to determine the applicability condition for a functional relation. To determine the functional relation applicable to a chosen literal (an fd-predicate) in the inference, we find the set of arguments (with arguments positions in increasing order) which are ground terms in the literal and with this a key, we traverse trie-structure and get set of dependent arguments. This forms the functional relation applicable to that instance of the literal.

3.4.5 Inference

Data-Structure

The interpreter uses an explicit stack. An iterative algorithm for inference is implemented, so that at any step complete proof-tree is accessible for the purpose of debugging etc. Each enrty in the stack corresponds to the activation record which contains the following information for example qlist, all facts and rules for the chosen literal, association list containing variables and their bindings, active and passive dependency information. These informations are represented using a list which is defined using defstruct as follows :

```
(defstruct (entry (:constructor make-entry (qlist
                                            facts
                                            rules
                                            alist
                                            afd
                                            pfd)))
          qlist
          facts
          rules
          alist
          afd
          pfd )
```

The advantage in defining an entry using defstruct is that it provides us with constructor and selector functions for making an entry and accessing different components of it.

Algorithm

```
;* INPUT to the interpreter is a list of literals i.e. qlist
;* which is to be solved.
;* qlist is renamed and the literal (next) is added at the en
;* this marks the end of the proof.
;* INITIALISATION OF THE STACK
;* in the begining alist a-fd-list p-fd-list are nil.
(let ((ini-stack (make-entry qlist
                             (all-facts (first-of query))
                             (all-rules (first-of query))
                             nil
                             nil
                             nil)))
;* TOP-LEVEL ITERATION
;* initialisation of loop variables
(do* ((stack ini-stack)
      (top-stack (first-of stack) (first-of stack))
      (qlist-lcl (entry-qlist top-stack)
                 (entry-qlist top-stack))
      (alist-lcl (entry-alist top-stack)
                 (entry-alist top-stack))
      (a-fd-list (entry-afd top-stack)
                 (entry-afd top-stack))
      (p-fd-list (entry-pfd top-stack)
                 (entry-pfd top-stack))
      (query (subst-variables (first-of qlist))
             (subst-variables (first-of qlist)))
      (over-flag nil))
((or (null stack) over-flag)
   ;* TERMINATION condition
```

```
(cond (over-flag
       ;* qlist is successful
       t)
      (t
       ;* qlist is unsuccessful
       '$fail-infer)))

;* body of top-level loop

(cond ((null qlist-lcl)

       ;* Sub-problem is solved. This arises when the argument
       ;* of negation is successful.

       (setq over-flag t))
      (t
       ;* Inference cycle with the top-stack begins here.
       ;* Search for a solution is made in depth-first
       ;* manner. First literal in the qlist is selected
       ;* as query to be solved.

       (cond ((its a non-database query)

              ;* query is a non-database query.
              ;* it could be any of the following :
              ;* not, =, computepred, procpred
              ;* cut, next etc.

              (let ((res (solve query)))
                   (cond ((not (fail-p res))

                          ;* query succeeds then

                          (modify-top-stack))
                         (t

                          ;* query fails so backtrack

                          (pop-out-top-stack)))))
             (t

              ;* Its a database query.
              ;* A clause needs to be chosen for solving
              ;* the query.

;* ITERATION FOR CHOOSING A CLAUSE

(do* ((clauses-to-try (get-clauses top-stack)
                      (get-clauses top-stack))
      (clause (rename-variables (first-of clauses))
              (rename-variables (first-of clauses)))
      (lcl-res (unify query (asrt-conseq clause) alist-lcl)
               (unify query (asrt-conseq clause) alist-lcl))
      ($contra-flag nil)
      ($bt-pred nil)
      ($bt-flag nil))

     ((or (null clauses-to-try) $contra-flag)

        ;* TERMINATION condition for iteration.
        ;* Either query has failed (i.e. no clauses found
        ;* for the query) or contradictory information
```

```
    ;* is found w.r.t. some fd-information.
    ;* BACKTRACK either naively or in a restricted
    ;* manner .
    (cond ($bt-flag (perform restricted backtracking))
          (t (perform naive backtracking))))
(cond ((unification succeeded)
       (cond ((query is an fd-predicate with non-nil
               dependent arguments)
              (cond ((unification instantiates only
                      dependent arguments )
                     ;* restrict query against backtrackin
                     (modify top-stack)
                     (push new top-stack))
                    (t
                     ;* Usual backtracking
                     (modify top-stack)
                     (push new top-stack))))
             (t
              ;* query either not an fd-predicate
              ;* or nil dependent arguments
              (modify top-stack)
              (push new top-stack))))
       (t
        ;* Unification has failed.
        (cond ((active-dependencies present)
               (cond ((failure violates
                           active-dependencies)
                      (set backtracking point)))))
        (cond ((query is fd-predicate with non-nil.
                dependent arguments)
               (cond ((failure due to dependent argument:
                       (cond ((failure due to
                               dependent arguments only)
                              ;* contradictory informatio
                              ;* is present
                              (setq $contra-flag t))))
                      (t 'try-rest-of-the-rules)))
              (t 'try-rest-of-the-rules)))))))))))
```

3.5 Conclusion

We have provided a facility for specifying functional relation information in logic programming and developed an interpreter which makes use of such information. We found that with the help of such facility user can eliminate unnecessary recomputation of goal. Further the specification of fd-information is not dependent on the inference strategy used by the interpreter but rather it is global and declarative in nature.

REFERENCES

[CM79] Clark K.L.,McCabe F.G., "The Control Facilities of IC-PROLOG" in Expert System in Micro Electronics Age, Michie D. (editor), University Of Edinburg, Scotland, 1979.

[CM84] Clocksin W.F., Mellish C.S., "Programming in PRO-LOG", Springer Verlag, (2nd edition) New York, 1984.

[D80] Davis R., "META-RULE : Reasoning About Control AI, vol 15, no 3, December 1980.

[DL84] Dincbas M., Le Pape J.P., " Meta-Control of Logic Program in METALOG " International Conference On Fifthe Generation Computer System, November 1984.

[GL82] Gallaire H., Lassere C., "A Control Meta-Language For Logic Programming", Logic Programming, Clark and Tarlund (eds), Academic Press, 1982.

[K74] Kowalski R.A., "Predicate Logic As A Programming Language " , IFIP, 1974.

[K79] Kowalski R., " Algorithm = Logic + Control CACM, vol 15, no 3, December 1979.

[KM87] Kumar A., Malhotra V.M., " A Look Ahead Inter-preter For PROLOG ", M.Tech. Thesis, Jan 1987, Department Of Computer Science, IIT Kanpur.

[L84] Lloyd J.W., "Foundation Of Logic Programming ", Springer Verlag, 1984.

[N86] Nakamura K., " Control Of Logic Program Execution Based On Functional Relation ", 3rd International Conference On Logic Programming, LNCS 225 ,1986.

[S86] Sangal R., " VIDHI : An Expert System Shell", TRCS-86-3Ø, Technical Report, Department Of Computer Science and Engineering, IIT Kanpur.

[SS86] Shapiro E., Sterling L., " The Art Of PROLOG ", The MIT Press, Series In Logic Programming, 1986.

APPENDIX 1

Reference Manual For VIDHI

VIDHI, a logic programming shell with some facilities which are specially suitable for developing an expert-system, is developed at IITK. A summary of commands for defining assertions, issuing goal etc are given here.

Defining Assertions

For defining assertions following commands are given

(defassert <asrt-name> <conseq> <- <antec1> ... <antecn>)

If literals in the antecedent of an assertion are absent then it defines a fact otherwise it defines a rule.

Each literal in the antecedent or consequent of an assertion looks like

(<predicate-name> <arg1> ... <argn>)

Posing Query

A query is asked by specifying the literals in the following way

(goal <literal1> ... <literaln>)

Remark : For defining question rules computational predicates, defproc predicates one may refer to VIDHI manual and VIDHI tutorial.

Functional Relations Information

The functional relation information corresponding to an fd-predicate, in the interpreter descibed in chapter 3, is specified as

(deffd (pred-name <arg1> <arg2> .... <argn>))

where each <argi> could be any of the following

i : if <argi> is an independent argument

d : if <argi> is a dependent argument
x : if <argi> is neither an independent nor a dependent argument.

When fd-information is specified then before posing a query user must compile the fd-information by giving the command

(fd-compile)

Meta-Control

For defining meta-clauses same syntactic framework is used. In the consequent position of a meta-clause only meta-predicate should appear. The details (syntax and semantics) of meta-predicates is described in chpter 2.

Trace and debug

The user can give a command
(debug)
which will print the stack in each inference cycle. For seeing the contents of stack, in each inference cycle, in case of meta-control (interpreter described in chapter 2) user must give the command
(mdebug)

The trace facility is implemented in meta-control interpreter only. The user can set the trace in the inference by giving following command
(settrace)
It will print GOAL, CLAUSE and RESOLVENT in each inference cycle. For resetting the trace user can give the following command
(resetmtrace)

APPENDIX 2

TRACE OF THE EXAMPLES

Example 2_1

object-clauses

```
(defassert ex2-1 (nxt red blue))
(defassert ex2-2 (nxt red green))
(defassert ex2-3 (nxt red yellow))
(defassert ex2-3 (nxt blue red))
(defassert ex2-4 (nxt blue green))
(defassert ex2-5 (nxt blue yellow))
(defassert ex2-6 (nxt green red))
(defassert ex2-7 (nxt green blue))
(defassert ex2-8 (nxt green yellow))
(defassert ex2-9 (nxt yellow red))
(defassert ex2-1Ø (nxt yellow green))
(defassert ex2-11 (nxt yellow blue))
(defassert ex2-12 (colour ?x1 ?x2 ?x3 ?x4 ?x5)
                  <- (nxt ?x1 ?x2)
                     (nxt ?x1 ?x4)
                     (nxt ?x1 ?x5)
                     (nxt ?x2 ?x3)
                     (nxt ?x2 ?x4)
                     (nxt ?x3 ?x4)
                     (nxt ?x4 ?x5))
```

meta-control

```
(defassert mex2-1 (activate ?r (nxt ?x ?y))
                  <- (totalinst (nxt ?x ?y)))
```

Execution Begins

```
(goal (colour ?x1 ?x2 ?x3 ?x4 ?x5))
   GOAL : (colour ?x11
                  ?x21
                  ?x31
                  ?x41
                  ?x51)
   CLAUSE : ((colour ?x14
                     ?x24
                     ?x34
                     ?x44
                     ?x54) <--
             ((nxt ?x14 ?x24)
              (nxt ?x14 ?x44)
              (nxt ?x14 ?x54)
```

```
                (nxt ?x24 ?x34)
                (nxt ?x24 ?x44)
                (nxt ?x34 ?x44)
                (nxt ?x44 ?x54)))
RESOLVENT : ((colour ?x11
                     ?x21
                     ?x31
                     ?x41
                     ?x51))

GOAL : (nxt ?x14 ?x24)
CLAUSE : ((nxt red blue))
RESOLVENT : ((nxt ?x14 ?x24)
             (nxt ?x14 ?x44)
             (nxt ?x14 ?x54)
             (nxt ?x24 ?x34)
             (nxt ?x24 ?x44)
             (nxt ?x34 ?x44)
             (nxt ?x44 ?x54))

GOAL : (nxt red ?x44)
CLAUSE : ((nxt red blue))
RESOLVENT : ((nxt red ?x44)
             (nxt red ?x54)
             (nxt blue ?x34)
             (nxt blue ?x44)
             (nxt ?x34 ?x44)
             (nxt ?x44 ?x54))

GOAL : (nxt blue blue)
GOAL HAS FAILED
RESOLVENT : ((nxt red ?x54)
             (nxt blue ?x34)
             (nxt blue blue)
             (nxt ?x34 blue)
             (nxt blue ?x54))

GOAL : (nxt red ?x44)
CLAUSE : ((nxt red green))
RESOLVENT : ((nxt red ?x44)
             (nxt red ?x54)
             (nxt blue ?x34)
             (nxt blue ?x44)
             (nxt ?x34 ?x44)
             (nxt ?x44 ?x54))
             (nxt green ?x54))
             (nxt green blue))

- - - - - - -

- - - - - - -

- - - - - - -

GOAL : (nxt blue ?x34)
CLAUSE : ((nxt blue red))
RESOLVENT : ((nxt blue ?x34) (nxt ?x34 green))
```

```
- - - - - - -
- - - - - - -
- - - - - - -

GOAL : (nxt blue ?x32)
CLAUSE : ((nxt blue red))
RESOLVENT : ((nxt blue ?x32)
             (nxt blue blue)
             (nxt ?x32 blue)
             (nxt blue blue))

GOAL : (nxt blue blue)
GOAL HAS FAILED
RESOLVENT : ((nxt blue blue) (nxt red blue) (nxt blue blue))

GOAL : (nxt red ?x42)
CLAUSE : ((nxt red green))
RESOLVENT : ((nxt red ?x42)
             (nxt red ?x52)
             (nxt blue ?x32)
             (nxt blue ?x42)
             (nxt ?x32 ?x42)
             (nxt ?x42 ?x52))

- - - - - - - -
- - - - - - - -
- - - - - - - -

GOAL : (nxt red green)
CLAUSE : ((nxt red green))
RESOLVENT : ((nxt red green) (nxt green blue))

GOAL : (nxt green blue)
CLAUSE : ((nxt green blue))
RESOLVENT : ((nxt green blue))

?x1=red
?x2=blue
?x3=red
?x4=green
?x5=blue
yes
should i try for another answer : n
ok
```

Example 2_4

object clauses

```
(defassert a1     (fa a b))
(defassert a2     (fa a c))
(defassert a3     (fa b d))
(defassert a4     (fa b e))
(defassert a5     (fa d h))
(defassert a6     (fa d i))
(defassert a7     (fa c f))
(defassert a8     (fa c g))
```

```
(defassert a9      (fa f j))
(defassert a10     (fa f k))
(defassert r1      (gfa ?x ?y)
                    <- (fa ?x ?z)
                       (fa ?z ?y))
```

meta-control

```
(defassert m1      (activate ?r (fa ?x ?y))
                    <- (atom ?y))
```

Execution Begins

```
(goal (gfa ?x k))

   GOAL : (gfa ?x1 k)
   CLAUSE : ((gfa ?x4 ?y4) <--
              ((fa ?x4 ?z4) (fa ?z4 ?y4)))
   RESOLVENT : ((gfa ?x1 k))

   GOAL : (fa ?z4 k)
   CLAUSE : ((fa f k))
   RESOLVENT : ((fa ?x4 ?z4) (fa ?z4 k))

   GOAL : (fa ?x4 f)
   CLAUSE : ((fa c f))
   RESOLVENT : ((fa ?x4 f))

?x=c
yes
should i try for another answer : n
ok
(goal (gfa c ?x))

   GOAL : (gfa c ?x1)
   CLAUSE : ((gfa ?x4 ?y4) <--
              ((fa ?x4 ?z4) (fa ?z4 ?y4)))
   RESOLVENT : ((gfa c ?x1))

   GOAL : (fa c ?z4)
   CLAUSE : ((fa c f))
   RESOLVENT : ((fa c ?z4) (fa ?z4 ?y4))

   GOAL : (fa f ?y4)
   CLAUSE : ((fa f j))
   RESOLVENT : ((fa f ?y4))

?x=j
yes
should i try for another answer : n
ok
```